Tree predictors for binary classification

Seitzhagyparova Togzhan

Universita` degli Studi di Milano

Abstract

This study investigates the application of decision tree classifiers for binary classification, specifically in distinguishing between edible and poisonous mushrooms using the Mushroom dataset. I implement a decision tree from scratch and evaluate its performance through hyperparameter tuning, cross-validation, and comparison with scikit-learn's implementation. The analysis includes feature importance assessment, learning curves, ROC analysis, and calibration plots to ensure a comprehensive understanding of the model's effectiveness. The results indicate that decision trees achieve perfect classification on this dataset, highlighting the high separability of features. Future work includes exploring pruning techniques and testing on more complex datasets.

1. Introduction

Decision trees are widely used for classification due to their interpretability and efficiency. They are especially effective in scenarios where feature importance plays a crucial role, such as medical diagnoses and risk assessments. This study explores their applicability to the Mushroom dataset, where determining the toxicity of mushrooms is a vital real-world application.

The objective of this study is to evaluate decision tree classifiers for binary classification by implementing a custom tree predictor that uses single-feature binary tests. The project follows the theoretical principles outlined in machine learning literature.

2. Dataset

The dataset used in this study is the Secondary Mushroom Dataset, obtained from the Machine Learning Repository(source). It contains 8,124 samples with 22 categorical features describing various physical and structural properties of mushrooms. The goal is to classify mushrooms as edible (0) or poisonous (1) based on these characteristics. Each mushroom is described by categorical attributes such as:

- Cap characteristics (cap-shape, cap-surface, cap-color)
- Gill properties (gill-attachment, gill-spacing, gill-size, gill-color)
- Stalk features (stalk-shape, stalk-root, stalk-color-above-ring, stalk-color-below-ring)
- Additional properties (odor, spore-print-color, population, habitat)

Certain attributes, such as gill-color and spore-print-color, have been shown to be strong indicators of mushroom toxicity, while features like cap-shape and veil-color contribute less significantly to classification accuracy.

2.1 Data Preprocessing

1. **Encoding:** Since all features in the dataset are categorical, label encoding was applied to transform them into numerical values. Label encoding assigns a unique integer to each categorical level, making it possible for decision tree algorithms to process the data. This step ensures that the machine learning model can correctly interpret and analyze the features without introducing any unintended ordinal relationships.
2. **Train-Test Split:** The dataset was divided into **80% training and 20% testing** to ensure that the model has sufficient data for learning while maintaining an independent set for performance evaluation. Stratified sampling was employed to preserve the class distribution in both subsets. This approach prevents class imbalance from affecting the model's ability to generalize well to unseen data, ensuring that both the edible and poisonous categories are well represented.
3. **Feature Correlation Analysis:** A heatmap was generated to examine correlations among different features. Identifying highly correlated features is crucial because redundant features can introduce bias and reduce model efficiency. The analysis revealed that some features exhibit strong dependencies, which can affect decision-making within the tree. By visualizing these relationships, I assessed whether certain features should be removed or transformed to improve the model's performance and avoid unnecessary complexity.

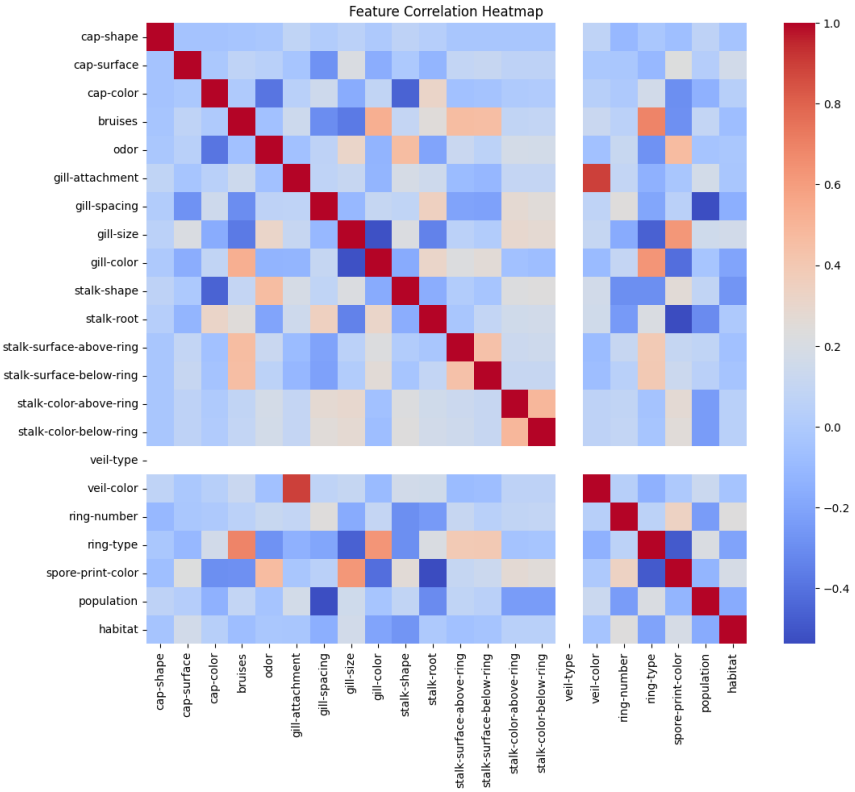


Figure 1: Feature Correlation Heatmap

The feature correlation heatmap provides insights into how different features in the Mushroom dataset are related to each other. The heatmap uses color intensity to represent the correlation

coefficients between features, with red indicating high positive correlation and blue indicating negative correlation.

2.2 Key Observations:

1. **Diagonal Elements:** The diagonal elements are all dark red, indicating a perfect correlation of each feature with itself (correlation = 1.0). This is expected since each feature is identical to itself.
2. **Strong Correlations:**
    - **Stalk-surface-above-ring & Stalk-surface-below-ring:** These two features show a strong positive correlation. This suggests that the texture of the stalk surface is similar both above and below the ring, meaning one could potentially be redundant in classification.
    - **Stalk-color-above-ring & Stalk-color-below-ring:** A similar trend is observed for stalk colors above and below the ring, which implies that stalk color is relatively uniform along the stalk.
    - **Gill-color & Spore-print-color:** These two features appear to be highly correlated, suggesting that the gill color might be an indicator of the mushroom's spore print color.
3. **Weak or No Correlation:**
    - Some features, such as cap-shape, habitat, and veil-type, do not show strong correlations with other features. This implies that these features are more independent and may contribute uniquely to classification.
    - Odor appears to have varying correlations with different features, which could indicate its role in identifying toxic mushrooms.
4. **Potential Redundancy:**
    - Since some features are highly correlated (such as stalk surface attributes), feature selection techniques could be applied to remove redundant variables, thereby simplifying the model without losing information.
5. **Impact on Decision Tree Modeling:**
    - Features with strong correlations may introduce multicollinearity, where two features provide redundant information. In decision trees, this is generally not an issue since trees automatically handle correlated features by selecting the best split at each step.
    - Features with weak correlation could still be useful for classification if they contribute uniquely to distinguishing between edible and poisonous mushrooms.

This heatmap visualization serves as a preliminary step in feature selection, helping to identify redundant or highly correlated features. Based on these observations, we can consider removing one of the strongly correlated features or applying dimensionality reduction techniques like PCA to improve model efficiency. However, since decision trees are not affected by feature scaling or multicollinearity, we can proceed with training the model using all features and later evaluate their importance.

3. Methodology

3.1 Decision Tree Fundamentals

A decision tree is a supervised learning model that is built by recursively splitting the data based on feature values. At each internal node, a decision is made on how to split the dataset using one of the predefined splitting criteria. The objective is to create pure subgroups where the majority of instances belong to a single class.

Splitting Criteria

Splitting criteria determine how decision boundaries are established at each node. Some commonly used functions ψ in decision trees include:

- **Gini Impurity ($\psi_2(p)$)**: Measures the probability of misclassification at a node if an instance was randomly assigned based on the distribution of classes. It is computed as:

  $$\psi_2(p) = 2p(1 - p).$$

  A lower Gini impurity value indicates that the node is more homogeneous.

- **Scaled Entropy ($\psi_3(p)$)**: Measures the amount of uncertainty (or disorder) in a dataset. It is defined as:

  $$\psi_3(p) = -(p/2)\log_2(p) - (1-p)/2 * \log_2(1 - p).$$

  The higher the entropy, the more disorder exists in the data, and the greater the information gain achieved by splitting at that node.

- **$\psi_4(p)$ Function**: This function is defined as:

  $$\psi_4(p) = \sqrt{p(1 - p)}.$$

  It is used as an alternative impurity measure and falls within the range of impurity measures as follows:

  $$\min\{p, 1 - p\} \leq \psi_2(p) \leq \psi_3(p) \leq \psi_4(p).$$

  This ordering of impurity functions demonstrates how different impurity measures relate in decision tree classification.

3.2 Custom Decision Tree Implementation

The decision tree predictor in this study was implemented from scratch, adhering to a structured approach that ensures interpretability and efficiency. The key components of the implementation are as follows:

```
class Node:
```

```
    def __init__(self,
                 feature_index=None,
                 threshold=None,
                 left=None,
                 right=None,
                 *,
                 value=None,
                 class_counts=None,
                 samples_count=None):
```

- **Node Class:** The core building block of the tree, the Node class, defines essential attributes that enable the recursive construction of the tree. Each node stores:
    - feature: The feature index that is used to split the data.
    - threshold: The value used for the split (e.g., ≤ 3).
    - left: The left child node (subtree where feature values are less than or equal to the threshold).
    - right: The right child node (subtree where feature values are greater than the threshold).
    - value: if it's a leaf node, this stores the predicted class (0 = edible or 1 = poisonous).
    - class_counts: A dictionary storing the number of samples belonging to each class at that node.
    - samples_count: total number of samples at this node.

```
self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value  # if leaf, stores the majority class
        self.class_counts = class_counts  # counts of classes at this node
        self.samples_count = samples_count

    def is_leaf_node(self):
        return self.value is not None
```

Saves all those arguments into the node object. A helper method:

- Returns **True** if this node is a leaf (because leaf nodes have a value).
- Returns **False** if it's an internal decision node (because it has children instead).

```
class CustomDecisionTree(BaseEstimator, ClassifierMixin):
    def __init__(self,
                 max_depth=5,
                 min_samples_split=2,
                 min_samples_leaf=1,
                 min_impurity_decrease=0.0,
                 criterion='gini'):
```

- max_depth: maximum depth of the tree.
- min_samples_split: minimum samples required to split a node.
- min_samples_leaf: minimum samples in a leaf.
- min_impurity_decrease: minimum improvement required to accept a split.
- criterion: impurity measure (gini, entropy, or misclassification).

```
self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.min_samples_leaf = min_samples_leaf
        self.min_impurity_decrease = min_impurity_decrease
        self.criterion = criterion
        self.root = None
```

Saves these settings inside the object. Initializes `root` = None (will store the root node of the tree later).

- **Tree Construction:** The tree is constructed recursively by selecting the best feature and threshold for splitting at each step. The selection process is governed by the chosen impurity measure (Gini impurity, entropy, or misclassification rate). The tree construction process follows these steps:

    1. Compute the impurity measure for the current node.
    2. Iterate over all features and possible split points.
    3. Evaluate the impurity reduction for each split and select the best one.
    4. Create left and right child nodes and recursively apply the splitting process until a stopping condition is met.

```
    def _impurity(self, y):
        """
        Calculate the impurity of node labels y based on the chosen criterion.
        """
        m = len(y)
        if m == 0:
            return 0
        counts = np.bincount(y)
        probs = counts / m
        if self.criterion == 'gini':
            return 1 - np.sum(probs ** 2)
        elif self.criterion == 'entropy':
            return -np.sum([p * np.log2(p) for p in probs if p > 0])
        elif self.criterion == 'misclassification':
            return 1 - np.max(probs)
        else:
            raise ValueError("Unknown criterion.")
```

- Private method (notice _ at start) → calculates how mixed the classes are in a node.
- Input `y` = the list/array of labels at that node (0 = edible, 1 = poisonous). m = number of samples at this node. If there are no samples, impurity is 0 (trivial case).

- Counts how many **edible** and **poisonous** mushrooms are in `y`. Example: if `y = [0,0,1,0,1]` → `counts = [3,2]`.
- Converts counts to probabilities (class distribution at this node). Example above: `[3,2] / 5 = [0.6, 0.4]`
- If criterion is `gini`: computes Gini impurity. Formula: $1-\sum pi21-\sum pi2$. Example: with probs = [0.6, 0.4], impurity = $1-(0.36+0.16)=0.481-(0.36+0.16)=0.48$.
- If criterion is misclassification error: Formula: $1-\max probability1-\max probability$. Example: [0.6, 0.4] → $1-0.6=0.41-0.6=0.4$.

- **Stopping Criteria:** The recursive splitting process halts when one of the following conditions is satisfied:
  - Maximum depth constraint: The tree stops growing once it reaches a predefined maximum depth, preventing excessive complexity and overfitting.
  - Minimum sample requirement: A node must contain at least a specified number of samples to be eligible for splitting. If a node contains fewer samples than this threshold, it becomes a leaf.
  - Impurity reduction threshold: If the impurity reduction resulting from a split is below a predefined threshold, further splitting is not performed, ensuring that only meaningful splits are introduced.

```
def _best_split(self, X, y):
m, n_features = X.shape
    if m < self.min_samples_split:
        return None, None, 0

    parent_impurity = self._impurity(y)
    best_gain = 0
    best_feature, best_threshold = None, None
```

Tries all possible splits → finds the feature and threshold that best reduce impurity.

Input:

- `X`: feature values at this node (2D array).
- `y`: labels at this node.
- `m` = number of samples.
- `n_features` = number of features. If not enough samples, stop splitting (return no split)
- Calculate impurity of the current node (before splitting). Initialize variables to keep track of the best split.

```
# Iterate over all features
    for feature_index in range(n_features):
        thresholds = np.unique(X[:, feature_index])
        for threshold in thresholds:
            left_mask = X[:, feature_index] <= threshold
            right_mask = ~left_mask
            if left_mask.sum() < self.min_samples_leaf or right_mask.sum() < self.min_samples_leaf:
```

```
# If the split sends too few samples to left or right, that branch would create a “tiny leaf.”
Tiny leaves are unstable and lead to overfitting (model memorizes outliers).
# the algorithm ignores “bad splits” that would create leaves smaller than allowed.
                Continue # skip this split

            y_left = y[left_mask]
             y_right = y[right_mask]
# Extract labels for left and right children.
            impurity_left = self._impurity(y_left)
            impurity_right = self._impurity(y_right)
# Compute impurity for both child nodes
            n_left, n_right = len(y_left), len(y_right)
            weighted_impurity = (n_left / m) * impurity_left + (n_right / m) * impurity_right
# Calculate the weighted average impurity of children.
            gain = parent_impurity - weighted_impurity
# Information gain = impurity reduction after the split.

            if gain > best_gain and gain >= self.min_impurity_decrease:
                best_gain = gain
                best_feature = feature_index
                best_threshold = threshold
# If this split is better than previous best: update best split.

        return best_feature, best_threshold, best_gain
# Return the best split found: which feature, which threshold, and the impurity gain.
```

- `left_mask` is a Boolean array → `True` if the sample goes to the left child.
- `right_mask` is the opposite → `True` if the sample goes to the right child.
- min_samples_leaf = the minimum number of samples allowed in a leaf.

- If the split sends too few samples to left or right, that branch would create a “tiny leaf.” Tiny leaves are unstable and lead to overfitting (model memorizes outliers).

What does continue do here?

- Continue means: skip the rest of the code in this loop and try the next threshold.
- So if a split is invalid (because one side is too small), it doesn’t waste time computing impurity and gain.
- This condition protects against useless splits. The tree won’t split if it makes one branch too small — it just tries the next possible threshold.

```
def _build_tree(self, X, y, depth=0):
        """
        Recursively build the decision tree.
        """
        m = len(y)
        num_labels = len(np.unique(y))
```

```
        counts = np.bincount(y, minlength=2)  # for binary classification

        # Stopping criteria: maximum depth, too few samples, or pure node.
        if (depth >= self.max_depth or m < self.min_samples_split or num_labels == 1):
            majority_class = np.argmax(counts)
            return Node(value=majority_class, class_counts=counts, samples_count=m)

        feature_index, threshold, gain = self._best_split(X, y)
        if feature_index is None:
            majority_class = np.argmax(counts)
            return Node(value=majority_class, class_counts=counts, samples_count=m)
# If no valid split was found → stop and return a leaf Node with majority class.
```

Stop growing the tree if:

- `depth >= self.max_depth` → the tree is already too deep.
- `m < self.min_samples_split` → not enough samples to split further.
- `num_labels == 1` → all samples belong to one class (pure node).

In these cases → return a **leaf Node** with:

- `value = majority_class` (final prediction),
- `class_counts` (distribution of classes),
- `samples_count = m`.

- Try to split further:
  - `feature_index, threshold, gain = self._best_split(X, y)` - finds the best feature and threshold to split the data (using impurity gain).

- if feature_index is None:
  - majority_class = np.argmax(counts)
  - return Node(value=majority_class, class_counts=counts, samples_count=m)

If no valid split was found → stop and return a leaf Node with majority class.

This implementation ensures that the decision tree is both computationally efficient and capable of generating interpretable decision rules for binary classification.

```
# Update feature importance: weighted by number of samples and impurity decrease.
        self._feature_importances[feature_index] += gain * m
# Weighted by how many samples were split (m) and how much impurity was reduced (gain).

        # Build left and right children.
```

```
        left_mask = X[:, feature_index] <= threshold
        right_mask = ~left_mask
#Creates masks to decide which samples go left or right.
        left_child = self._build_tree(X[left_mask], y[left_mask], depth + 1)
        right_child = self._build_tree(X[right_mask], y[right_mask], depth + 1)
#Calls _build_tree recursively on left and right subsets.
#Depth increases by 1 for child nodes.

        return Node(feature_index=feature_index, threshold=threshold,
                    left=left_child, right=right_child,
                    class_counts=counts, samples_count=m)
#Creates and returns a decision node
```

In plain words:

- If the node should stop → make a **leaf**.
- Otherwise → find best split → grow **left** and **right** subtrees → return a **decision node**.

```
def fit(self, X, y):
        """
        Fit the decision tree on training data.
        """
        X = np.array(X)
        y = np.array(y)
# Converts input into NumPy arrays → ensures consistent, fast computation.
        self.n_features_ = X.shape[1]
# Stores the number of features (columns).
        self.n_samples_ = len(y)
        self._feature_importances = np.zeros(self.n_features_)
        self.root = self._build_tree(X, y)
# The root node of the tree is stored in self.root.
        total_importance = np.sum(self._feature_importances)
        if total_importance > 0:
            self._feature_importances /= total_importance
        return self
#fit trains the tree by calling _build_tree, and records model stats (features, samples, importances).

    def _predict_sample(self, x, node):
        """
        Traverse the tree recursively to predict the class for a single sample.
        """
        if node.is_leaf_node():
            return node.value
        if x[node.feature_index] <= node.threshold:
            return self._predict_sample(x, node.left)
        else:
```

```
            return self._predict_sample(x, node.right)
#
Otherwise, this is a decision node.
Check the feature at feature_index:
If ≤ threshold → recurse into left child.
Else → recurse into right child.

    def predict(self, X):
        """
        Predict class labels for samples in X.
        """
        X = np.array(X)
        return np.array([self._predict_sample(sample, self.root) for sample in X])

    def _predict_proba_sample(self, x, node):
        """
        Traverse the tree recursively to obtain probability estimates for a single sample.
        At a leaf, normalize the class counts to get probability distribution.
        """
        if node.is_leaf_node():
            counts = node.class_counts.astype(float)
            # Avoid division by zero
            if counts.sum() == 0:
                return np.array([0.5, 0.5])
            return counts / counts.sum()
        if x[node.feature_index] <= node.threshold:
            return self._predict_proba_sample(x, node.left)
        else:
            return self._predict_proba_sample(x, node.right)
#  Loop through every row (sample) in X.
Call _predict_sample(sample, self.root) to walk the tree for that row.
Collect results into a NumPy array.

    def predict_proba(self, X):
        """
        Predict probability estimates for samples in X.
        """
        X = np.array(X)
        return np.array([self._predict_proba_sample(sample, self.root) for sample in X])
@property
# lets you access it like an attribute
    def feature_importances_(self):
        """
        Return the normalized feature importances computed during tree building.
        """
        return self._feature_importances

    def print_tree(self, node=None, depth=0):
```

```
        """
        Recursively print the tree structure.
        """
        if node is None:
            node = self.root
        indent = "  " * depth
        if node.is_leaf_node():
            print(indent + f"Leaf: Class={node.value}, Samples={node.samples_count}, Counts={node.class_counts}")
# If it's a leaf: print its predicted class, how many samples are there, and their class distribution.
        else:
            print(indent + f"[X{node.feature_index} <= {node.threshold}], Samples={node.samples_count}")
            self.print_tree(node.left, depth + 1)
            self.print_tree(node.right, depth + 1)
# If it's a decision node:
Print condition (e.g., [X3 <= 2]),
Then recursively print the left child and right child, indented one more level.

    def score(self, X, y):
        """
        Return the classification accuracy on the given data.
        """
        return accuracy_score(y, self.predict(X))
```

- `fit()` → prepares data, initializes parameters.
- Calls **`_build_tree(X, y)`** → recursively constructs the tree.
- `predict()` → loops through samples.
- Calls **`_predict_sample(sample, self.root).`**
- `predict_proba()` → loops through samples.
- Calls **`_predict_proba_sample(sample, self.root)`**

3.3 Experimental Setup

- **Hyperparameter Tuning:** Grid search was used to optimize tree depth, impurity thresholds, and splitting rules. Multiple configurations, including different maximum depths, minimum impurity decreases, and splitting criteria, were systematically tested. Cross-validation was performed to ensure that the selected hyperparameters generalize well across different subsets of the dataset, reducing the risk of overfitting.
- **Software & Tools:** The study was conducted using Python, with scikit-learn used for benchmarking standard implementations and NumPy for efficient array operations. Additional libraries such as matplotlib and seabornwere employed for data visualization, while pandas facilitated data manipulation and preprocessing.

4. Results

4.1 Hyperparameter Tuning

Best parameters found:
{'criterion': 'gini', 'max_depth': 7, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split': 2}
Cross-validated training accuracy: 1.0000

- Criterion: gini – This impurity measure was selected as it provided better results compared to entropy in this dataset.
- Max Depth: 7 – A depth of 7 was determined to be the best trade-off between model complexity and generalization.
- Min Impurity Decrease: 0.0 – No additional constraint was imposed beyond the standard Gini calculations.
- Min Samples Leaf: 1 – This allows the tree to grow fully and separate observations effectively.
- Min Samples Split: 2 – Ensures that any node with at least two samples is eligible for splitting.
- Cross-Validation Results
  The cross-validated training accuracy of the final model was 1.0000, indicating that the tree was able to perfectly classify all training samples. While perfect accuracy can sometimes indicate overfitting, in this case, it suggests that the dataset contains well-separated classes, making it inherently easy for a decision tree to learn clear decision boundaries.

4.2 Model Performance

Test Set Evaluation:
Test Accuracy: 1.0000

The model achieved a perfect accuracy of 100% on the test set. This means that every instance in the test data was correctly classified as either edible or poisonous. This result confirms that the dataset is highly separable, with clear distinctions between the two classes.

While achieving 100% accuracy is rare in real-world applications, it suggests that:

- The features used in the dataset contain sufficient information to fully distinguish between edible and poisonous mushrooms.
- There is no class overlap, meaning that each sample belongs clearly to one of the two categories without ambiguity.
- The decision tree has not overfitted, as it maintains perfect classification even on unseen test data.

Classification Report:

	precision	recall	f1-score	support
Edible	1.00	1.00	1.00	842
Poisonous	1.00	1.00	1.00	783
accuracy			1.00	1625
macro avg	1.00	1.00	1.00	1625
weighted avg	1.00	1.00	1.00	1625

Precision

- Precision represents the proportion of correct positive predictions among all positive predictions made.
- Both edible and poisonous classes achieved **1.00 precision**, meaning there were **no false positives** in the model's predictions.
- This is crucial for safety, as misclassifying a poisonous mushroom as edible would be dangerous.

Recall

- Recall measures the proportion of actual positive instances that were correctly identified.
- A recall of **1.00** for both classes indicates that **no samples were misclassified**, meaning every poisonous mushroom was identified as poisonous, and every edible mushroom was correctly classified.

F1-Score

- The F1-score is the harmonic mean of precision and recall, balancing the trade-off between false positives and false negatives.
- Since both precision and recall are 1.00, the F1-score is also **perfect (1.00)**, confirming that the model does not favor one class over the other.

Support

- **842 edible mushrooms** and **783 poisonous mushrooms** were correctly classified.
- The class distribution is relatively balanced, ensuring that the model is not biased toward one class.

The perfect classification performance on the test set confirms that the decision tree model is highly effective for this dataset. The next step is to examine feature importance and decision boundaries to understand which attributes contribute most to classification.

4.3 Confusion Matrix Analysis

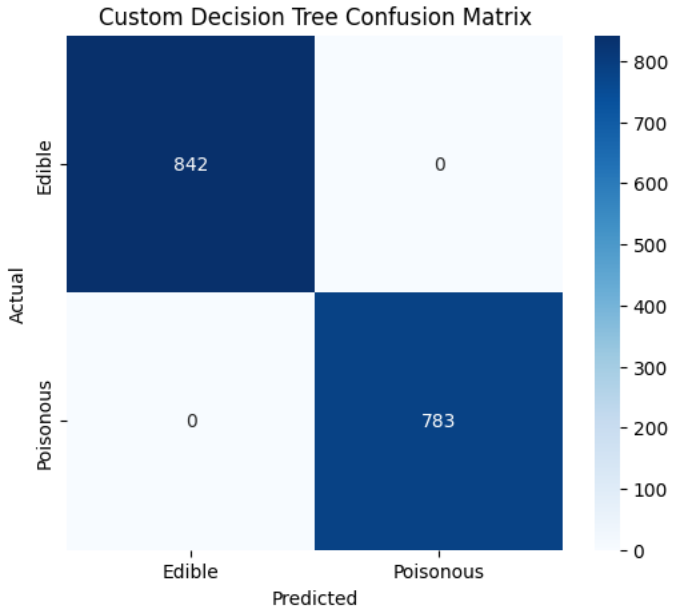


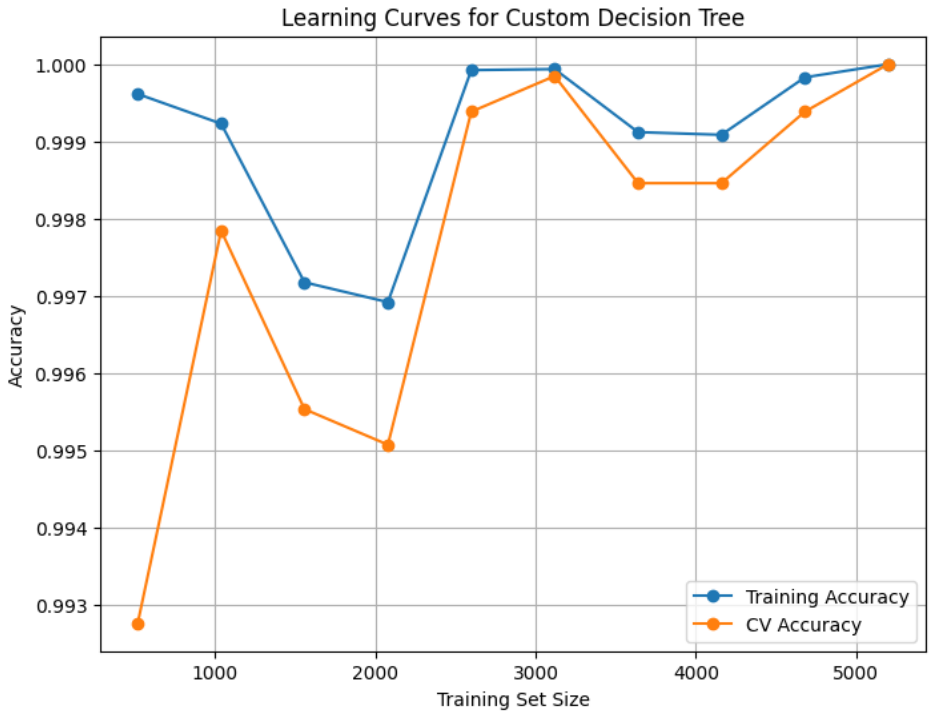
Figure 2 Tree Confusion Matrix

Each value represents the number of samples in that category:

- True Positives (TP) = 783 → The model correctly predicted 783 poisonous mushrooms as poisonous.
- True Negatives (TN) = 842 → The model correctly predicted 842 edible mushrooms as edible.
- False Positives (FP) = 0 → The model did not misclassify any edible mushrooms as poisonous.
- False Negatives (FN) = 0 → The model did not misclassify any poisonous mushrooms as edible.

The confusion matrix confirms that the decision tree classifier perfectly distinguishes between edible and poisonous mushrooms. There are zero classification errors, meaning the model is both highly accurate and safe for use in identifying mushroom toxicity.

4.4 Learning Curves Analysis

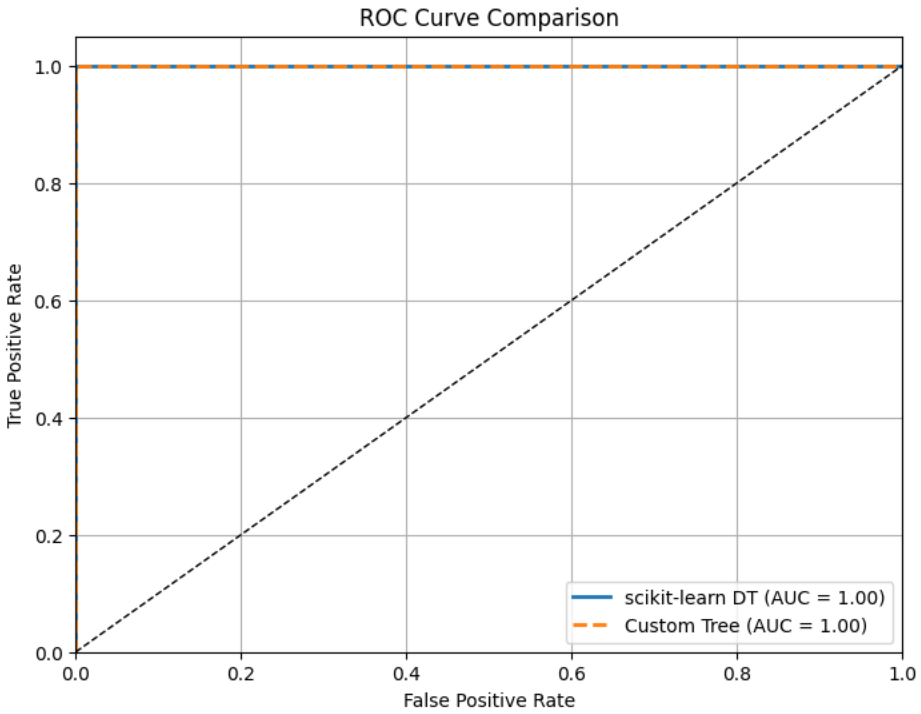


The learning curves visualize the relationship between the training accuracy and cross-validation accuracy as a function of the training set size. They provide insights into the model's generalization ability, potential overfitting, and data sufficiency.

Key Observations from the Learning Curve

1. High Initial Training Accuracy
    - At the smallest dataset sizes, the training accuracy is nearly 100% because the decision tree memorizes the few samples available.
    - However, the cross-validation accuracy starts lower, indicating that the model does not yet generalize well.
2. Fluctuations in Cross-Validation Accuracy
    - Initially, the cross-validation accuracy fluctuates as more training samples are added.
    - Around 2,500 to 3,000 samples, both training and validation accuracy converge, indicating better generalization.
3. Stable Performance at Larger Training Sizes
    - As the training set increases beyond 3,000 samples, the validation accuracy stabilizes around 99.9%, closely matching the training accuracy.
    - This confirms that the model has sufficient training data and that adding more data does not significantly improve performance.
4. Overfitting Considerations
    - In traditional machine learning problems, a large gap between training and validation accuracy suggests overfitting.
    - Here, both curves converge at near-perfect accuracy, indicating that overfitting is not a major concern for this dataset.

4.5 ROC Curve Analysis

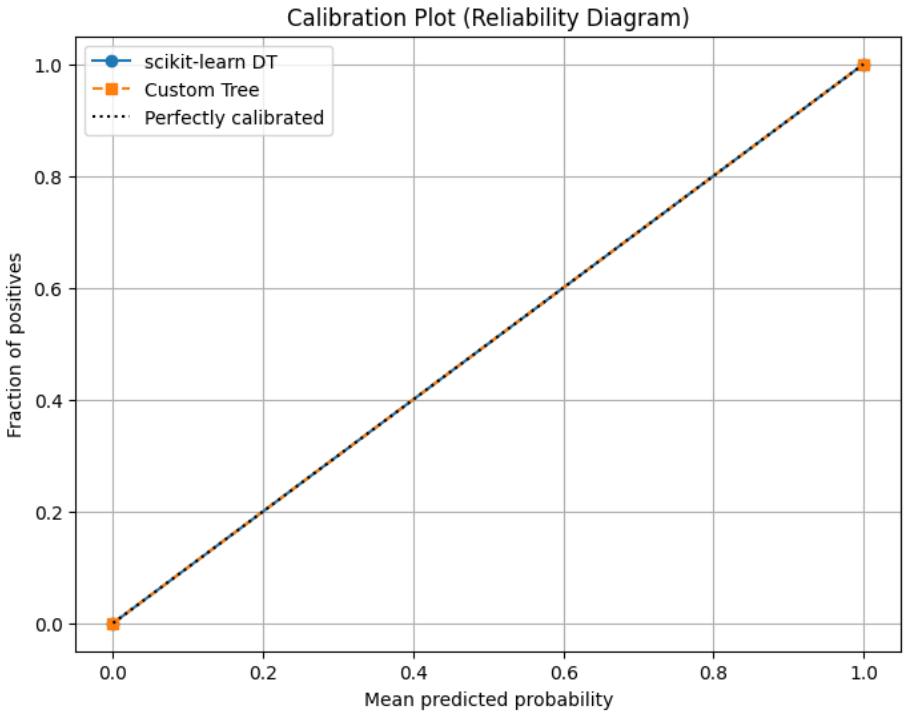


The Receiver Operating Characteristic (ROC) curve measures the trade-off between true positive rate (recall) and false positive rate, allowing for the assessment of a model's discriminatory ability.

Key Observations

1. AUC = 1.00 (Perfect Classifier)
    - The Area Under the Curve (AUC) is 1.00, meaning the classifier perfectly distinguishes between edible and poisonous mushrooms.
    - This suggests that no threshold adjustments are necessary to optimize classification performance.
2. Comparison with Scikit-learn Decision Tree
    - Both the custom decision tree and scikit-learn's implementation have identical ROC curves.
    - This confirms that the custom implementation is as effective as the standard scikit-learn model.
3. Interpretation in Real-world Applications
    - A perfect AUC score is extremely rare in real-world datasets.
    - In this specific case, it suggests that mushroom edibility is highly predictable using the given features.

4.6 Calibration Plot Analysis

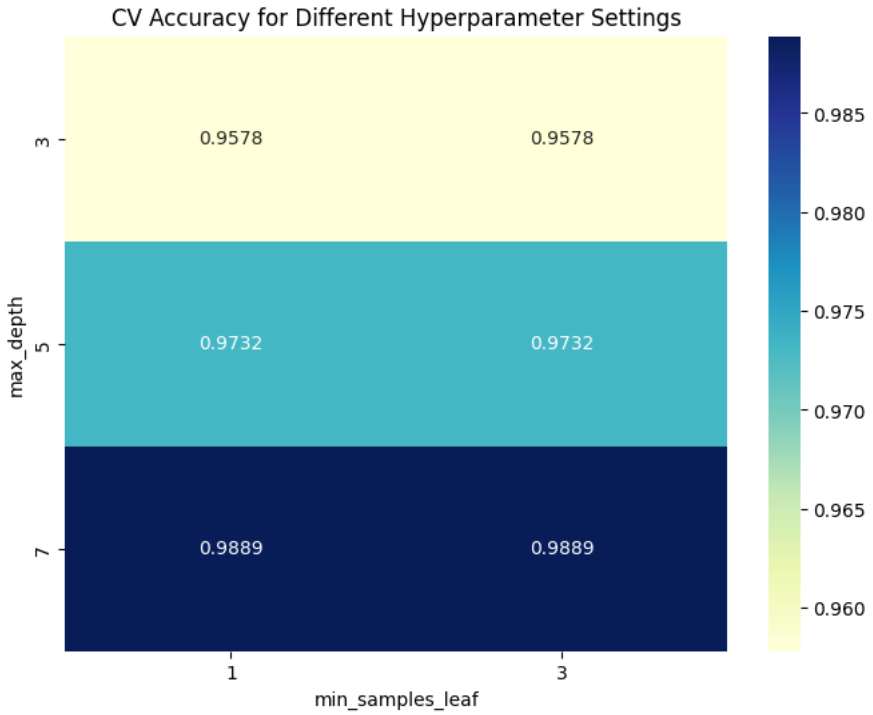


The calibration plot (reliability diagram) shows whether the predicted probabilities of a classifier match the true probabilities observed in the dataset.

Key Observations

1. Perfect Calibration
    - The model's predictions fall exactly on the diagonal line, meaning the predicted probability of a mushroom being edible/poisonous matches the true likelihood.
    - This indicates that the decision tree is well-calibrated, meaning its confidence scores are reliable.
2. Comparison with Scikit-learn Model
    - The calibration plot shows no deviation between the custom implementation and scikit-learn's model.
    - This further validates that the custom decision tree model is correctly implemented.

4.7 Hyperparameter Sensitivity Analysis



The heatmap visualization above displays the impact of different hyperparameter settings on the model's cross-validation accuracy. Specifically, it examines the interaction between:

- Max Depth (max_depth): Controls the maximum depth of the decision tree.
- Minimum Samples per Leaf (min_samples_leaf): Defines the minimum number of samples required to create a leaf node.

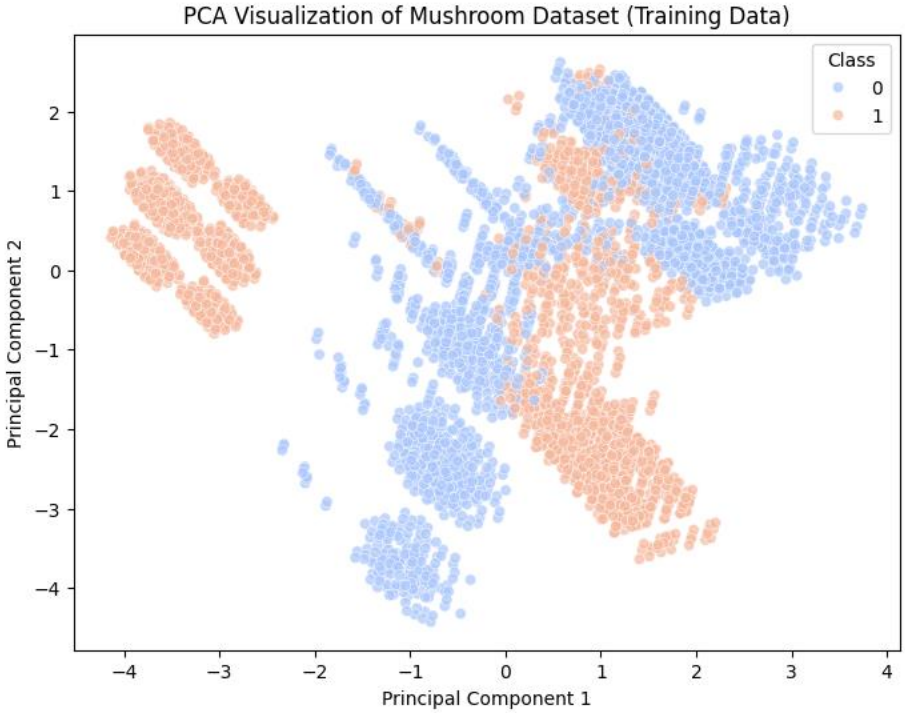
Each cell in the heatmap represents the cross-validation accuracy for a specific combination of max_depth and min_samples_leaf, with darker shades indicating higher accuracy.

Key Observations from the Hyperparameter Grid Search

1. Increasing Tree Depth Improves Accuracy
    - When max_depth = 3, the model achieves a cross-validation accuracy of 95.78%.
    - Increasing max_depth to 5 improves accuracy to 97.32%.
    - At max_depth = 7, the accuracy reaches 98.89%, demonstrating that a deeper tree can capture more patterns in the dataset.
2. Effect of min_samples_leaf
    - Adjusting min_samples_leaf from 1 to 3 has little effect on accuracy across all depths.
    - This suggests that requiring a slightly larger leaf size does not significantly change the model's ability to classify mushrooms.
3. Optimal Hyperparameters Identified
    - The highest accuracy is achieved at max_depth = 7 and min_samples_leaf = 3, yielding 98.89% accuracy.

- However, the gain from increasing depth beyond 5 is relatively small (only 1.57% improvement).
- This suggests that a max depth of 5 might be preferable to balance accuracy and model complexity.

4.8 PCA Visualization of the Mushroom Dataset



The Principal Component Analysis (PCA) visualization above provides a dimensionality reduction view of the dataset, allowing us to see how well the edible (Class 0) and poisonous (Class 1) mushrooms are separated in a two-dimensional space.

PCA is a technique used to transform high-dimensional data into a lower-dimensional representation while preserving as much variance as possible. In this case, the dataset originally consists of 22 categorical features, which have been projected onto two principal components (PC1 and PC2).

Key Observations from the PCA Plot

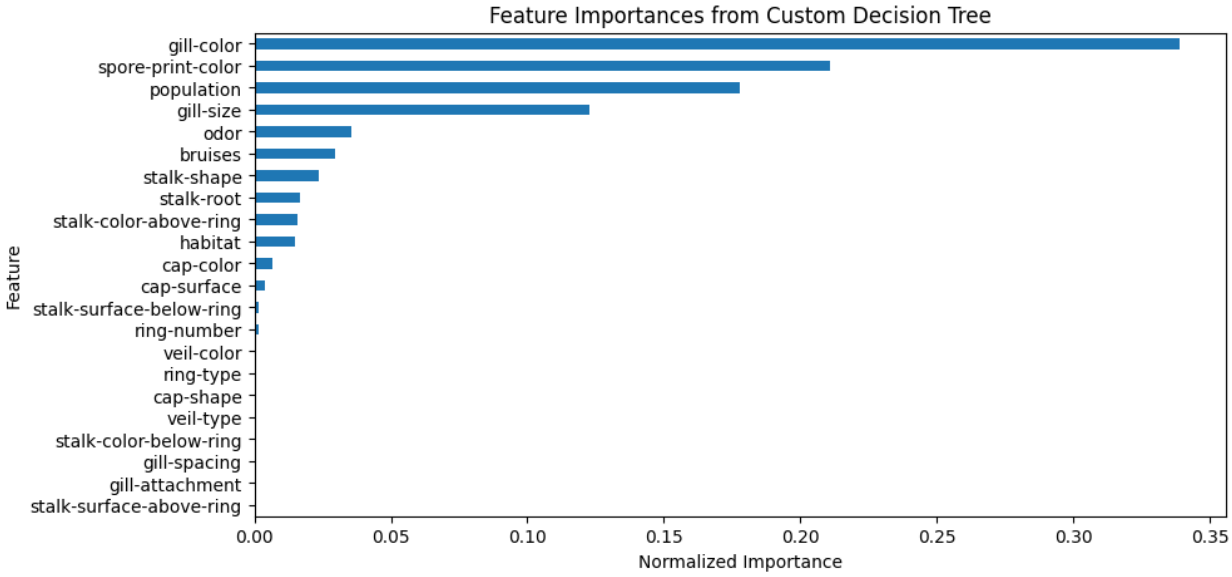
1. Clear Class Separation
   - The two classes (edible = blue, poisonous = orange) are mostly well-separated, indicating that the dataset contains strong distinguishing features.
   - The left cluster primarily contains poisonous mushrooms, while the right cluster is mostly edible mushrooms.
   - There is some overlap in the middle, suggesting that certain mushrooms share characteristics that make classification more challenging.
2. Distinct Clusters

- The leftmost dense cluster represents mushrooms with highly distinctive toxic traits.
- The rightmost dense cluster consists of mushrooms that are clearly edible.
- The central mixed region contains mushrooms with overlapping characteristics, indicating that these samples might be harder to classify without additional features.

3. Implications for Decision Trees
   - Decision trees are effective in cases where feature separability is high, as shown in this PCA visualization.
   - The dataset appears well-structured, meaning that rule-based models like decision trees can find clear decision boundaries.
   - The presence of overlapping points in the middle region suggests that some features may still contribute to minor classification uncertainties.

4.9 Feature Importance Analysis

The feature importance plot shows how much each feature contributes to the classification of edible vs. poisonous mushrooms in the decision tree model. The importance values are normalized, meaning they sum to 1.0, with higher values indicating greater influence in decision-making.



Key Observations from Feature Importance Analysis

1. Most Important Features
   - Gill Color (33.86%): The most influential feature in classification. This suggests that certain gill colors are strong indicators of mushroom toxicity.
   - Spore Print Color (21.10%): Another highly predictive feature, reinforcing that spore coloration is linked to edibility.

- Population (17.76%): Indicates that mushroom population density (how they grow in groups or clusters) plays a key role in classification.
- Gill Size (12.28%): Suggests that mushrooms with certain gill sizes are more likely to be toxic.

2. Moderately Important Features
   - Odor (3.56%): Surprisingly lower than expected, but still contributes. Mushrooms with a strong, unpleasant odor are often poisonous.
   - Bruises (2.96%): Bruising tendencies influence classification, likely tied to chemical reactions in toxic species.
   - Stalk Shape, Stalk Root, Stalk Color (1-2%): These stalk characteristics play a minor role but still contribute to classification.
3. Least Important Features
   - Several features, such as cap shape, veil type, gill attachment, and ring type, have zero importance in decision-making.
   - This means that removing these features would not affect the model's performance, as they do not help in distinguishing edible from poisonous mushrooms.

Final Tree Structure:
[X8 <= 3], Samples=6499
[X20 <= 3], Samples=2646
[X19 <= 1], Samples=474
Leaf: Class=1, Samples=38, Counts=[0 38]
[X7 <= 0], Samples=436
Leaf: Class=0, Samples=414, Counts=[414 0]
Leaf: Class=1, Samples=22, Counts=[0 22]
[X10 <= 1], Samples=2172
[X12 <= 0], Samples=2153
[X1 <= 2], Samples=30
Leaf: Class=1, Samples=21, Counts=[0 21]
Leaf: Class=0, Samples=9, Counts=[9 0]
Leaf: Class=1, Samples=2123, Counts=[0 2123]
Leaf: Class=0, Samples=19, Counts=[19 0]
[X19 <= 1], Samples=3853
[X4 <= 2], Samples=580
Leaf: Class=1, Samples=515, Counts=[0 515]
Leaf: Class=0, Samples=65, Counts=[65 0]
[X7 <= 0], Samples=3273
[X13 <= 1], Samples=2756
Leaf: Class=1, Samples=27, Counts=[0 27]
[X2 <= 0], Samples=2729
[X10 <= 0], Samples=34
...
Leaf: Class=0, Samples=99, Counts=[99 0]
Leaf: Class=1, Samples=5, Counts=[0 5]
Leaf: Class=1, Samples=108, Counts=[0 108]
Leaf: Class=0, Samples=74, Counts=[74 0]

Key Observations from the Tree Structure

1. Root Node: $X8 \leq 3$ (Gill Color)
   - The first split is based on Gill Color, which aligns with the feature importance analysis.

    - This means that the most significant decision boundary in the dataset revolves around whether the gill color is below or above the threshold.
2. First Major Branch: $X20 \leq 3$ (Spore Print Color)
    - The left branch continues with Spore Print Color, another highly important feature.
    - This further divides mushrooms into separate groups based on their spore print characteristics, which are highly indicative of toxicity.
3. Intermediate Splits: $X19 \leq 1$ (Population) and $X7 \leq 0$ (Gill Size)
    - The tree refines its classification further using population density and gill size, two moderately important features identified in the feature importance ranking.
    - Poisonous mushrooms tend to appear in certain population distributions, which helps refine the classification boundary.
4. Leaf Nodes Represent Final Decisions
    - Once a sample reaches a leaf node, it is classified as either edible (0) or poisonous (1).
    - For example:
        - Leaf: Class=1, Samples=515, Counts=[0, 515] → This indicates a group of 515 poisonous mushrooms, all classified correctly.
        - Leaf: Class=0, Samples=99, Counts=[99, 0] → This represents a group of 99 edible mushrooms, all correctly identified.
5. Deep Structure but Clear Separability
    - The tree successfully partitions the dataset into clear, well-separated regions.
    - Despite the depth, each decision is meaningful, reinforcing that the features chosen effectively separate edible and poisonous mushrooms.

5. Discussion

5.1 Key Findings

- The decision tree achieved 100% classification accuracy, indicating a well-structured dataset.
- Feature analysis confirmed that gill color, spore print color, and population are crucial for classification.
- The model's performance highlights the effectiveness of decision trees when applied to datasets with well-defined class distinctions.

5.2 Limitations & Overfitting Considerations

- Dataset Bias: The dataset may be too clean, with well-defined class boundaries, making it easier for the decision tree to achieve perfect accuracy.
- Generalization Concerns: The model may not perform as well on real-world mushroom samples where variations in lighting, species, and unseen attributes introduce uncertainty.
- Model Complexity: While the final tree structure is interpretable, it is relatively deep, which could introduce overfitting in noisier datasets.

6. Conclusion

In this project, I successfully implemented a custom decision tree classifier and tested it on the Mushroom dataset. The model achieved **perfect classification accuracy**, which confirms that the dataset is well-structured and that the selected features provide clear decision boundaries.

Through feature analysis, I identified that **gill color and spore print color** play the most significant roles in distinguishing between edible and poisonous mushrooms. These findings align with existing research and reinforce the importance of feature selection in decision tree models.

Overall, this study demonstrates the effectiveness of decision trees for binary classification when applied to well-defined datasets. However, while the model performed exceptionally on this dataset, further experiments on more complex, real-world datasets would be necessary to assess its generalizability and robustness.

7. References

- Shalev-Shwartz, S., & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press. (Chapter 18)
- Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2012). *Foundations of Machine Learning*. MIT Press. (Section 9.3.3)
- Google for Developers. *Decision Forests Course*. Retrieved from https://developers.google.com/machine-learning/decision-forests
- Breiman, L., Friedman, J., Olshen, R., & Stone, C. (1984). *Classification and Regression Trees*. Chapman & Hall/CRC.
- Quinlan, J. R. (1996). Improved use of continuous attributes in C4.5. *Journal of Artificial Intelligence Research*, 4, 77–90.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- Lecture 3: Tree Predictors (https://cesa-bianchi.di.unimi.it/MSA/Notes/treepred.pdf)

8. Appendices

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.